

A Simple Algorithm for Subgraph Queries in Big Graphs

Chemseddine Nabti* and Hamida Seba*

*Université de Lyon, CNRS, Université Lyon 1
LIRIS, UMR5205, F-69622 Lyon, France.
hamida.seba@univ-lyon1.fr

March 17, 2017

Abstract

Subgraph queries also known as subgraph isomorphism search is a fundamental problem in querying graph-like structured data. It consists to enumerate the subgraphs of a data graph that match a query graph. This problem arises in many real-world applications related to query processing or pattern recognition such as computer vision, social network analysis, bioinformatic and big data analytic. Subgraph isomorphism search knows a lot of investigations and solutions mainly because of its importance and use but also because of its NP-completeness. Existing solutions use filtering mechanisms and optimise the order within which the query vertices are matched on the data vertices to obtain acceptable processing times. However, existing approaches are iterative and generate several intermediate results. They also require that the data graph is loaded in main memory and consequently are not adapted to large graphs that do not fit into memory or are accessed by streams. To tackle this problem, we propose a new approach based on concepts widely different from existing works. Our approach distills the semantic and topological information that surround a vertex into a simple integer. This simple vertex encoding that can be computed and updated incrementally reduces considerably intermediate results and avoid to load the entire data graph into main memory. We evaluate our approach on several real-world datasets. The experimental results show that our approach is efficient and scalable.

1 Introduction

Graphs are not a new paradigm for data representation and modeling. Their use in these domains dates back to the birth of computer databases with, for example, the work of Bachman on the *Network* database model [4]. However, the advent of applications related to nowadays almost fully connected world with social networks, online crime detection, genome and scientific databases,

etc., has brought graphs to greater prominence. This is due mainly to their adaptability to represent the linked aspect of nowadays data but also to their flexibility and scalability when dealing with the main challenge of these kind of applications : Massive data. In this context, subgraph isomorphism search is a fundamental task on which search and querying algorithms are based. Subgraph isomorphism search also known as exact subgraph matching or subgraph queries is the problem of enumerating all the occurrences of a query graph within a larger graph called the data graph (cf. Figure 1). Subgraph isomorphism search is an

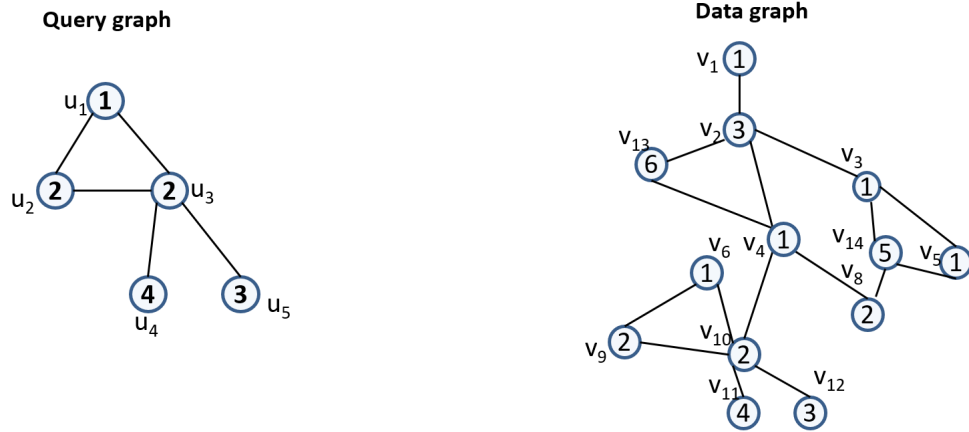


Figure 1: Running Example.

NP-complete problem that knows extensive investigations and solutions mainly because of its importance and use, but also because known solutions are memory and space expensive and consequently are not adapted for big graphs [8, 12, 14]. Most existing solutions are extensions of the well known Ullman’s algorithm [15]. These solutions are based on exploring a search space in the form of a recursion tree that maps the query vertices to the data graph’s vertices. However, parsing the recursion tree is exponential in function of the number of vertices in the involved graphs. So, existing solutions never construct entirely the recursion tree and use pruning methods to have smaller search spaces. Detailed solutions are surveyed in several papers [8, 12, 14]. The most referenced methods such as Ullman’s algorithm [15], VF2 [7], QuickSI [13], GraphQL [11], GADDI [16], SPath [17] and Turbo_{ISO} [9] have different approaches to tackle this problem. Two main pruning mechanisms are used by existing methods:

1. The order within which query vertices are matched to the data graph’s vertices: In fact, this order impacts directly the size of the recursion tree. The most efficient vertex ordering is based on the frequency of labels [18]. Handling vertices having infrequent vertex labels and infrequent adjacent edge labels as early as possible will reduce the size of the search space drastically. However, when these frequencies are obsolete, they do not produce

relevant orders. This leads to larger search spaces and consequently slower isomorphism verification algorithms.

2. The k -neighborhood of query vertices: this is the amount of information used when matching a query vertex with data vertices. The more information is used, i.e., k is big, the more the pruning of the search space can be global. However, representing compactly the k -neighborhood for practical comparisons is a challenging issue.

Moreover, these backtracking-based solutions keep the entire data graph into memory for adjacency verification and do not adapt to graphs that do not fit into memory. In this paper, we aim to reduce the search space and keep as few data as possible in main memory to address the problem of finding efficiently all the occurrences of a query graph in a big data graph. Our contributions are:

- We propose a novel encoding of vertices that distills all the information around a vertex in a single integer leading to a simple but extremely efficient filtering scheme for processing subgraph isomorphism search. The whole filtering process is based on integer comparisons.
- Our encoding mechanism has the advantage to adapt to all graph access models: main memory, external memory and streams. By performing one sequential scan of the disk file (or the stream of edges) of the input graph, we avoid expensive random disk accesses if the graph do not fit into memory.
- We present a new subgraph isomorphism search framework integrating the proposed vertex encoding. This algorithm significantly reduces the isomorphism verification costs and produces a maximally reduced search space.
- We conduct extensive experiments using public datasets in different application domains to attest the effectiveness and efficiency of the proposed scheme.

The rest of this paper is organized as follows. Section 2 first formalizes the problem of subgraph isomorphism search and defines the notations used throughout the paper, then, it discusses related work to motivate our contribution. In Section 3, we introduce our main contribution: the vertex neighborhood encoding and how it is used to solve subgraph isomorphism search. Section 4 presents a comprehensive experimental studies on several datasets. Section 5 concludes the paper.

2 Background

2.1 Problem Definition

We use a data graph to represent objects and their relationships using vertices and edges. More formally, A data graph G is a 3-tuple $G = (V, E, \ell)$, where V is a set of vertices (also called nodes), $E \subseteq V \times V$ is a set of edges connecting the vertices, $\ell : V \rightarrow \Sigma$ is a labeling function on the vertices and the edges where Σ is the set of labels that can appear on the vertices and/or the edges. We denote the vertex set and the edge set of a graph G as $V(G)$ and $E(G)$, respectively. We use $|V(G)|$ and $|E(G)|$ to represent respectively the number of vertices and the number of edges in G .

An undirected edge between vertices u and v is denoted indifferently by (u, v) or (v, u) . A neighbor of a vertex v is a vertex adjacent to v . The degree of a vertex is the number of its neighbors. We use $N(v)$ to represent the neighbors of vertex v . $\ell(u)$ represents the label of vertex u and $\ell((u, v))$ is the label of the edge (u, v) in G .

A graph that is contained in another graph is called a subgraph and can be defined as follows:

Definition 1. A graph $G_1 = (V_1, E_1, \ell_1)$ is a subgraph of a graph $G_2 = (V_2, E_2, \ell_2)$ if $V_1 \subseteq V_2$, $E_1 \subseteq E_2$, $\ell_1(x) = \ell_2(x) \forall x \in V_1$, and $\ell_1(e) = \ell_2(e) \forall e \in E_1$.

Graph isomorphism is defined as follows:

Definition 2. A graph $G_1 = (V_1, E_1, \ell_1)$ and a graph $G_2 = (V_2, E_2, \ell_2)$ are said to be isomorphic if there exists a bijective function $h : V_1 \rightarrow V_2$ such that the following conditions are met:

1. $\forall x \in V_1 : \ell_1(x) = \ell_2(h(x))$
2. $\forall e \in E_1 : \ell_1(e) = \ell_2(h(e))$
3. $\forall (x, y) \in E_1 : (h(x), h(y)) \in E_2$
4. $\forall (h(x), h(y)) \in E_2 : (x, y) \in E_1$

Given a query graph Q and a data graph G , the subgraph isomorphism search of Q in G is the problem of enumerating all the subgraphs of G that are isomorphic to Q .

For presentation convenience, we focus on non-directed simple labeled graphs (graphs with no loops nor multiple edges) but our results can be straightforwardly extended to deal with directed graphs or non simple graphs. For the same reasons, we do not show edge labels on our examples but these labels are considered in our algorithms.

2.2 Related Work

Many algorithms are proposed to solve the subgraph isomorphism search problem. We can cite without being exhaustive Ullman’s algorithm [15], VF2 [7], QuickSI [13], GraphQL [11], GADDI [16], SPath [17], Turbo_{ISO} [9] and CFL-match [5]. One can find in [12] a meaningful study that surveys and compares most of these methods. In this section, we review the existing solutions on other facets that outline and justify our contributions. Existing algorithms for subgraph isomorphism search are built onto two basic tasks: Filtering and Verification. Filtering is the most important step and determines the efficiency of the algorithm. The verification step is generally based on the Ullman’s backtracking subroutine [15] that searches in a depth-first manner for matchings between the query graph and the data graph obtained by the filtering step. So, the aim of the filtering step is to reduce the search space on which the verification step operates. Filtering mechanisms can be classified into two categories depending on their scope: local or global. A local refinement mechanism prunes the set of mappings that are candidates for a single vertex. A global punning reduces globally the search space.

- Global pruning: as global punning mechanisms, we can cite vertex ordering and query rewriting. Vertex ordering is the selection of an order within which the vertices of the query are handled. In fact, this order has a direct incidence on the size of the search space as demonstrated by several examples [12]. Query rewriting consists in representing the query in a form that simplifies its matching. Ullman’s algorithm [15] and SPath do not define any global pruning mechanism and picks the query vertices in a random manner. VF2 and GADDI handle a query vertex only if it is connected to an already matched vertex. However, GADDI uses an additional mechanism : a distance based on the number of frequent substructures between the k -neighborhoods of two vertices as a mean to prune globally the search space after each established mappings between a query vertex and a data vertex. QuickSI rewrites the query in the form of a tree: a spanning tree of the query. Edges and vertices of the query are weighted by the frequency of their occurrence in the data graph. Based on these weights, a minimum spanning tree is constructed and used to search the data graph. GraphQL selects the vertex that minimises the cost of the ongoing join operation. The cost of a join is estimated by the size of the product of the involved sets of vertices. Turbo_{ISO} uses the ordering introduced in [18]. This ordering uses the popularity of query vertices in the data graph. Every query vertex u is ranked as follows [18]:

$$rank(u) = \frac{freq(G, \ell(u))}{deg(u)}$$

where $freq(G, l)$ is the number of data vertices in G that have label l , and $deg(u)$ means the degree of u . This ranking function favors lower frequencies and higher degrees which will minimize the number candidates vertices in the data graph. Furthermore, Turbo_{ISO} rewrites the query within a tree using this ranking as in QuickSI but Turbo_{ISO} aggregates

the vertices that have the same labels and the same neighbors into a single vertex. More recently, the authors of [5] claim that spanning trees are not the best solutions to represent queries. They show that the edges not included in the spanning tree may have an important pruning power. So, they propose to enhance the tree representation by partitioning the query graph into a core and a forest.

- Local pruning consists mainly in reducing the number of mappings available for each query vertex. In fact, the final search space is the result of joining the sets of available mappings of each query vertex. Thus, given a query graph $Q(V_Q, E_Q, \ell)$ and a data graph $G = (V_G, E_G, \ell)$, the aim is to reduce as much as possible the sets $C(u_i)$, $i = 0, |V_Q| - 1$, where $C(u_i)$ is the set of vertices of the data graph that match the query vertex u_i . The final search space is obtained by joining these sets, i.e., $C(u_1) \times C(u_2) \times \dots \times C(u_{|V_Q|-1})$ [11]. The reduction of $C(u)$ is generally achieved using the neighborhood information of u . The amount of the obtained pruning depends on the scope of the considered neighborhood. The simplest solution considers the one-hop neighborhood such as the degree of the vertex and/or the labels of the neighbors. Neighborhood at k -hops is also used in some methods. Ullman’s Algorithm refines C_u by removing the vertices that have a smaller degree than u . GraphQL also uses the direct neighborhood by encoding within a sequence the labels of the neighbors of each vertex. Furthermore, GraphQL uses an approximation algorithm proposed in [10] to further reduce the search space by discarding the data vertices that are not compatible with the query vertex using the k -neighborhood around u . VF2 looks to 2-hops neighborhood. SPath uses the k -neighborhood by maintaining for each vertex u a structure that contains the labels of all vertices that are at a distance less or equal than k from u . SPath uses its encoding of the k -neighborhood to remove the data vertices that have a k -neighborhood that does not englobe any k -neighborhood of query vertices. By rewriting, the query within a tree, QuickSI and Turbo_{ISO} uses also the k -neighborhood with the particularity that the neighborhood is rooted at a more pruning vertex. The tree representation of Turbo_{ISO} is also more compact as it aggregates similar vertices.

From the above discussion of existing methods, one can see that the main concern is reducing the search space. The pillar of such quest is the amount of semantic and topological information we maintain for each vertex and how this information is encoded. The more information we use the more pruning we achieve. However, the encoding of this information has a direct impact of its usefulness in pruning. Moreover, existing methods use static encoding of the vertex neighborhood in that it is not updatable after a local pruning. We think that a simple encoding that can be simply updated after each local pruning will produce a more important global pruning of the search space.

3 A Novel Approach

In this paper, we propose a novel approach to subgraph isomorphism search that aims to maximally reduce the search space. The method is also adapted for all access methods and especially for big graphs that are accessed within a stream of in external memory. The main task of the proposed framework is a filtering step that reduces the data graph to only the occurrences of the query graph. Then, a listing step parses these occurrences to report them. Contrarily to existing work, no matching phase is necessary. All the parts of the data graph that are kept in memory, after filtering, are occurrences of the query. So, there are no intermediate results nor unpromising branches to prune. The efficiency of the filtering step relies on a novel method to encode a vertex. This encoding distills all the neighboring information that characterise a vertex into a single integer that uniquely identifies the vertex within the graph. Unlike existing methods that statically and invariably encode neighboring information, the vertex encoding integer can be dynamically updated leading to an iterative filtering process that prune globally the search space.

In the following, we first describe this encoding method, called neighborhood encoding integer, then, we describe the filtering and listing steps of our subgraph matching framework.

3.1 Vertex Neighborhood Encoding Integer

In our method the high-level idea is to put in a simple integer the neighborhood information that characterise a vertex. Matching two vertices is then a simple comparison between integers. Given a vertex u , the neighborhood encoding of u , denoted $ne(u)$, distills the whole structure that surrounds the vertex into a single integer. It is the result of a bijective function that is applied on the vertex's neighborhood information. This function ensures that two given vertices u and v will never have the same neighborhood encoding if they have the same number of neighbors unless they are isomorphic. Let $x_1, x_2, x_3, \dots, x_k$ be the list of u 's neighbors' labels. The neighborhood encoding of u is given by:

$$ne(u) = h(1, x_1) + h(2, x_1 + x_2) + \dots + h(k, x_1 + x_2 + x_3 + \dots + x_k).$$

$$ne(u) = \sum_{j=1}^k h(j, x_1 + \dots + x_j) \text{ where } h(k, p) = \binom{k+p-1}{k} = \frac{(k+p-1)!}{k!(p-1)!}$$

Theorem 3 states that $ne(u)$ is a bijection.

Theorem 3. $\forall (x_1, x_2, x_3, \dots, x_k) \in \mathbb{N}^k$ and $k > 0$, g_k is a bijective function from \mathbb{N}^k in \mathbb{N} , where:

$$g_k(x_1, x_2, x_3, \dots, x_k) = \sum_{j=1}^k h(j, x_1 + \dots + x_j)$$

and

$$h(k, p) = \binom{k+p-1}{k} = \frac{(k+p-1)!}{k!(p-1)!}$$

To use this bijection on vertices' labels, we need to assign a unique integer to each vertex label. This assignment can be simply achieved by numbering labels

parting from 1. We denote $ord(\ell(u))$ the numbering used to obtain the integer associated to the label of vertex u .

To uniquely identify a vertex in a graph G , we use three numbers: the label of the vertex, its degree and its neighborhood encoding integer. These three numbers form a unique footprint that identifies the vertex in the graph.

It is worth noting that the label of the vertex can be included in the vertex encoding integer leading to a unique integer that identifies uniquely the vertex rather than using the three values: label, degree and the neighborhood encoding integer. However, as it will be revealed in the following, having these three values separately allows to filter data graphs that are not sorted, i.e., the edges are read randomly.

So, in the proposed approach, each vertex u is associated to a footprint $\mathfrak{f}(u)$ that can be computed incrementally. $\mathfrak{f}(u)$ contains the three numbers that identifies uniquely a vertex u in a graph G :

1. $\mathfrak{f}(u).label$: the label of u ,
2. $\mathfrak{f}(u).degree$: the degree of u , and
3. $\mathfrak{f}(u).ne$: the neighborhood encoding of u ,

For example, to compute the footprint of vertex u_2 of the query graph of our running example, we have: $\mathfrak{f}(u_2).label = 2$, $\mathfrak{f}(u_2).degree = 2$, and $\mathfrak{f}(u_2).ne = h(1, 1) + h(2, 1 + 2) = 3$

To use footprints for graph matching, we use the following definitions:

Definition 4. Let u be a query vertex and v be a data vertex. Let $\mathfrak{f}(u)$ and $\mathfrak{f}(v)$ be respectively the vertex footprints of vertices u and v . $\mathfrak{f}(u)$ matches $\mathfrak{f}(v)$ iff

- $\mathfrak{f}(u).label = \mathfrak{f}(v).label \wedge \mathfrak{f}(u).degree < \mathfrak{f}(u).degree \wedge \mathfrak{f}(u).ne < \mathfrak{f}(v).ne)$, or
- $(\mathfrak{f}(u).label = \mathfrak{f}(v).label \wedge \mathfrak{f}(u).degree = \mathfrak{f}(u).degree \wedge \mathfrak{f}(u).ne = \mathfrak{f}(v).ne)$.

Definition 5. A query vertex u is mapped to data vertex v iff $\mathfrak{f}(u)$ matches $\mathfrak{f}(v)$.

To enumerate all the subgraphs of a data graph G that are isomorphic to a query graph Q , we use vertex footprints. For each vertex of the query graph, we compute its corresponding footprint. We obtain the query footprint:

$$\mathfrak{F}(Q) = \{\mathfrak{f}(u), u \in V(Q)\}.$$

Algorithm 1 presents the pseudo-code for computing the footprint of the query graph. The algorithm takes as input a query graph and produces as output its footprint $\mathfrak{F}(Q)$ and the set of labels of the query vertices denoted $\mathcal{L}(Q)$.

Figure 2 illustrates the footprint for our query graph example.

The set $\mathcal{L}(Q)$ is used when dealing with the data graph as a first filter. It allows to identify the vertices of the data graph for which we can compute a footprint because they are more likely to be in a subgraph that is isomorphic to the query. This also helps to filter and discard rapidly the vertices for which it is not necessary to compute a footprint. In fact, the data graph is handled almost

Algorithm 1: Query processing.

Data: A Query Graph $Q = (V, E)$

Result: The Query's footprint, $\mathfrak{F}(Q)$ and the set $\mathcal{L}(Q)$ of labels present in Q .

```
begin
   $\mathfrak{F}(Q) \leftarrow \emptyset$ ;
  foreach  $u \in V(Q)$  do
     $L(Q) \leftarrow \ell(u)$ ;
     $\mathfrak{f}(u).label \leftarrow \ell(u)$ ;
     $\mathfrak{f}(u).degree \leftarrow \deg(u)$ ;
     $\mathfrak{f}(u).ne \leftarrow \sum_{i=1}^{\deg(u)} (\mathfrak{h}(i, \sum_{k=1}^i (\{ord(\ell(v)) | v \in N(u) \text{ and } ord(\ell(v)) < i\})))$ ;
     $\mathfrak{F}(Q) \leftarrow \mathfrak{F}(Q) \cup \mathfrak{f}(u)$ ;
  end
end
```

similarly to the query graph. However, the idea is to reduce intermediate results. So, we compute a footprint for vertices that have a label in $\mathcal{L}(Q)$ and we keep only the vertex footprints that match at least a footprint in the query graph. So, only the vertices of G that match vertices of Q according to Definitions 4 and 5 are stored. This set of vertices of the data graph is denoted $\mathcal{V}_Q(G)$. Algorithm 2 presents the actions performed to process the data graph and compute its footprints according to the query graph that is handled. In this algorithm, we assume that the edges of the data graph are not sorted. They arrive randomly without a predetermined order. So footprints are computed at the end of the stream of edges. However, we will see in the following subsection that if a sorting is available, the storage of these edges is not necessary and that we can prune drastically the data graph during its reading.

Figure 2 illustrates the footprint computed for our data graph according to the labels of the considered query graph. This figure shows that vertices v_{13} and v_{14} of the data graph are not considered in the footprint of the data graph: they do not match any query label. We also note that v_{13} and v_{14} are also not considered in the computation of the footprints of their neighbors, neither in the degree nor in the neighborhood encoding integer. In that respect, the degree of vertex v_2 is equal to 3 and its neighborhood encoding integer does not include the label of vertex v_{13} . Compared to Algorithm 1, Algorithm 2 is written to suit a graph stream processed edge by edge or a graph that cannot be loaded entirely in memory and that can be processed by blocks of edges.

3.2 Filtering step

The graph obtained after processing all the edges of the data graph contains only the data vertices that match a query vertex according to the label of the query vertices. The filtering step removes iteratively the footprints and conse-

Algorithm 2: Data Graph processing.

Data: A Data Graph G .

Result: $\mathfrak{F}_Q(G)$: the Data Graph's footprint for the query Q .

begin

$\mathfrak{F}_Q(G) \leftarrow \emptyset$;

$V_Q(G) \leftarrow \emptyset$;

 // processing of the vertices

foreach *vertex* $u \in V(G)$ **do**

if $\ell(u) \in \mathcal{L}(Q)$ **then**

$V_Q(G) \leftarrow V_Q(G) \cup \{u\}$;

$\mathfrak{f}(u).label \leftarrow \ell(u)$;

$\mathfrak{f}(u).degree \leftarrow 0$;

$\mathfrak{f}(u).ne \leftarrow 0$;

$\mathfrak{F}_Q(G) \leftarrow \mathfrak{F}_Q(G) \cup \mathfrak{f}(u)$;

end

end

 //processing of the edges

foreach *edge* $e = (u, v) \in E(G)$ **do**

if $u \in V_Q(G)$ *and* $v \in V_Q(G)$ **then**

$\mathfrak{f}(u).degree ++$;

$\mathfrak{f}(v).degree ++$;

$N(u) \leftarrow N(u) \cup \{v\}$;

$N(v) \leftarrow N(v) \cup \{u\}$;

end

end

foreach *vertex* $u \in V_Q(G)$ **do**

$\mathfrak{f}(u).ne \leftarrow \sum_{i=1}^{\mathfrak{f}(u).degree} (\mathfrak{h}(i, \sum_{k=1}^i (\{\text{ord}(\ell(v)) \mid v \in N(u) \text{ and } \text{ord}(\ell(v)) < i\})))$;

end

end

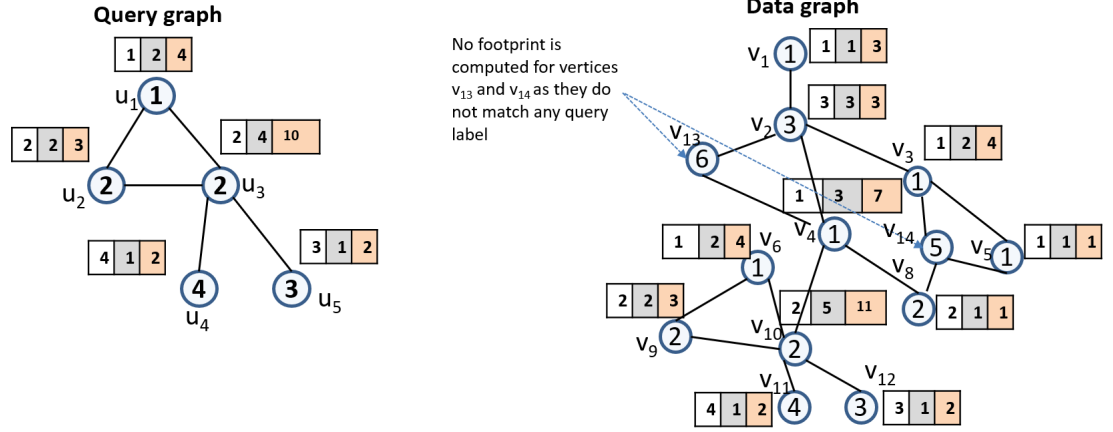


Figure 2: Footprints of the Query graph and the Data graph.

quently the vertices that do not match a query vertex using the degree and the vertex neighborhood encoding according to the matching condition stated by Definitions 4 and 5. Each time a vertex is removed by the filtering process the footprints of its neighbors are updated giving rise to new filtering opportunities. Algorithm 3 details the filtering step and Figure 3 illustrates this step on our example. Figure 3(a) shows the data graph filtered on labels during its reading. It shows also the other pruning operations that are available when using the other parts of the footprints. The first iteration of the filtering algorithm, finds out that vertices v_1 , v_5 and v_8 cannot be mapped to any query vertices. In fact, we can see that:

- According to the labels, only u_1 is a candidate to be mapped to v_1 and v_5 but according to the degree reported in the footprints, we can see that $f(u_1).degree > f(v_1).degree$ and $f(u_1).degree > f(v_5).degree$
- we have also for u_2 and u_3 , the vertices that have the same label as v_8 , $f(u_2).degree > f(v_8).degree$ and $f(u_3).degree > f(v_8).degree$

The mappings available at this point are:

- u_1 can be mapped to v_3 and v_6 ,
- u_2 can be mapped to v_9 and v_{10} ,
- u_3 can be mapped to v_{10} ,
- u_4 can be mapped to v_{11} , and
- u_5 can be mapped to v_2 and v_{12} .

So, v_1 , v_5 and v_8 can safely be removed from the set of vertices candidates to match the query graph. Note also that these pruning operations are done during

the computation of the footprints. As a result, the footprints of the neighbors of these removed vertices are updated: the degree and the vertex neighborhood encoding integer decrease for vertices v_2 , v_3 and v_4 leading to a new filtering iteration. The graph obtained after the first filtering iteration is illustrated by Figure 3(b). This second filtering iteration reveals that vertices v_3 and v_4 , that were mapped to u_1 during the first iteration, are no more mappable to this vertex. In fact, v_3 shows now a degree less than the degree of u_1 and v_4 that shows a same degree as u_1 has now a neighborhood encoding integer different from the neighborhood encoding integer of u_1 . So, v_3 and v_4 are removed from the filtered data graph depicted in Figure 3(c).

A third filtering iteration is then possible by removing v_2 which is no more mappable to u_5 as it has a lower degree.

The resulting final filtered data graph contains only the subgraph isomorphic to the query. It is depicted in Figure 3(d).

Note that at the end of this step, we have a set of candidates that exactly matches the query. No matching is further necessary because the structural verifications were encoded into the vertices' neighborhood encoding integers. So, the aim of the next step is just to put together the vertices that belong to the same solution, i.e., a subgraph that match the query, in order to report a comprehensive result. This step allows also to check edge labels as illustrated by Function *neighborCheck* (see Algorithm 7).

It is worth noting that if we have a small data graph, it can be processed in memory. For the other computation models such as external memory, the data graph can be read by blocks. For a graph stream, we process the graph edge by edge. For all these cases, we need a single parse of the graph. We suppose that we first read the vertices and their labels then we have the list of edges of the graph. The degrees of the vertices are computed incrementally while reading the edges. The vertex neighbourhood encoding integers are computed at the end of edges' reading. This is because, we suppose in the algorithm that the edges arrive in a random order. However, if the edges are sorted, The vertex neighborhood encoding integers can also be computed incrementally. In this case, the filtering process can be achieved during edges reading. Algorithm 4 gives the detailed actions to be performed to filter the footprints as edges are read. The impact of edges sorting is straightforward on our running example as we obtain the filtered data graph depicted on Figure 3(d) while reading the data graph edges. This is explained by the fact that all the filtering operations are achieved during the graph acquisition.

3.3 Subgraph Listing

After the filtering step, the data graph contains only the subgraphs that are isomorphic to the query. The subgraph listing step enumerate these subgraphs by parsing the final filtered data graph. All the vertices that are parsed in this step are relevant. So, no pruning rules or further filtering are necessary. Algorithm 5 implements this step. It puts together the vertices that are in the same occurrence of the query graph. The process is repeated until all the filtered

Algorithm 3: Data graph footprint exploration and filtering.

Data: $V_Q(G)$, Q , $\mathfrak{F}_Q(G)$ and $\mathfrak{F}(Q)$

Result: The subgraphs of G that are isomorphic to Q .

begin

 filter \leftarrow *FALSE*;

repeat

foreach vertex $v \in V_Q(G)$ **do**

if $\forall u \in V(Q), !FootMatch(\mathfrak{f}(u), \mathfrak{f}(v))$ **then**

 remove $\mathfrak{f}(v)$ from $\mathfrak{F}_Q(G)$;

 remove v from $V_Q(G)$;

foreach $x \in N(v)$ **do**

 remove v from $N(x)$;

$\mathfrak{f}(u).degree - -$;

 update $\mathfrak{f}(x).ne$;

end

 filter \leftarrow *TRUE*;

end

end

until !filter;

foreach vertex $u \in V(Q)$ **do**

$C(u) \leftarrow \{v \in V_Q(G) \text{ such that } FootMatch(\mathfrak{f}(u), \mathfrak{f}(v))\}$;

if $C(u) = \emptyset$ **then**

return (\emptyset) ;

end

end

$M \leftarrow \emptyset$;

 SubgraphList(M);

end

Algorithm 4: Data Graph processing and filtering with a sorted list of edges.

Data: A Data Graph G .

Result: $\mathfrak{F}_Q(G)$: a filtered Data Graph's footprint for the query Q .

begin

 //processing a stream of sorted edges

$u \leftarrow V_Q(G).firstVertex$;

while read edge $e = (x, v)$ from $E(G)$ **do**

while $x \neq u$ **do**

 ignore e ;

end

foreach $e = (u, v)$ **do**

if $v \in V_Q(G)$ **then**

$\mathfrak{f}(u).degree++$;

$\mathfrak{f}(v).degree++$;

$N(u) \leftarrow N(u) \cup \{v\}$;

$N(v) \leftarrow N(v) \cup \{u\}$;

$\mathfrak{f}(u).ne \leftarrow \mathfrak{f}(u).ne +$

$h(\mathfrak{f}(u).degree, \sum_{k=1}^{\mathfrak{f}(u).degree} (\{ord(\ell(y)) \mid y \in N(u)\}))$;

$\mathfrak{f}(v).ne \leftarrow \mathfrak{f}(v).ne +$

$h(\mathfrak{f}(v).degree, \sum_{k=1}^{\mathfrak{f}(v).degree} (\{ord(\ell(y)) \mid y \in N(v)\}))$;

end

end

if $\forall y \in V(Q), !FootMatch(\mathfrak{f}(y), \mathfrak{f}(u))$ **then**

 remove $\mathfrak{f}(u)$ from $\mathfrak{S}(G)$;

 remove u from $V_Q(G)$;

end

end

$u \leftarrow V_Q(G).nextVertex$;

end

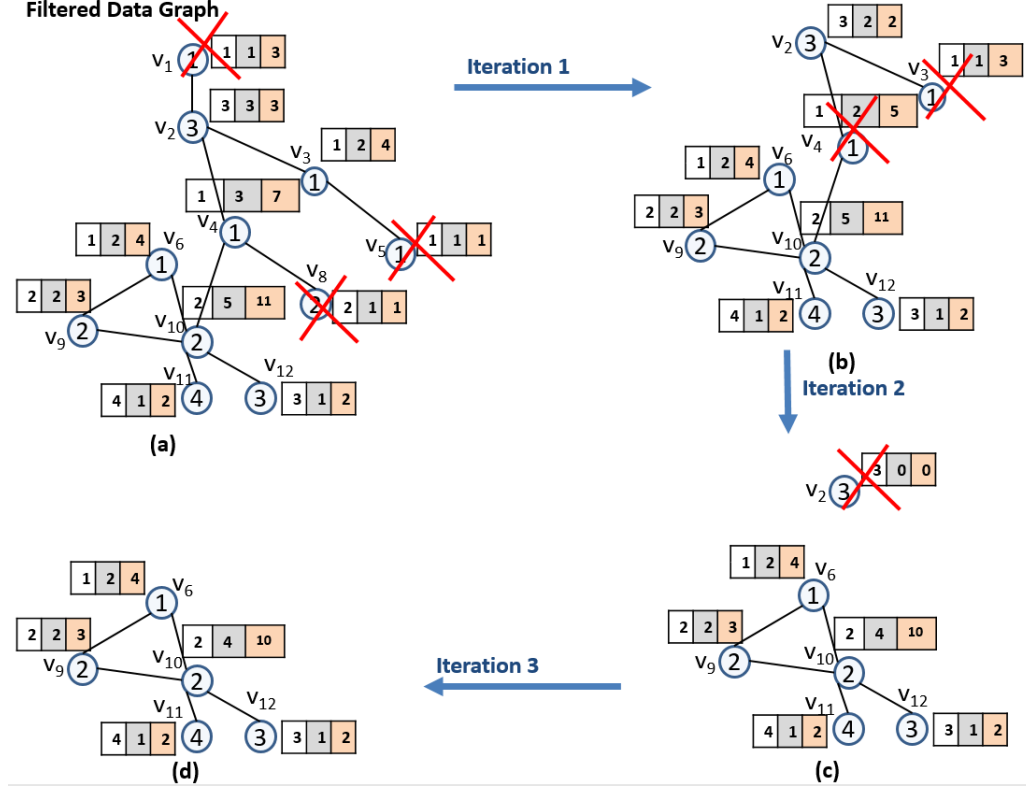


Figure 3: The filtering step.

graph is parsed. This step allows also to handles edge labels by discarding those that do not match the query labels. Algorithm 5 details subgraph listing. This algorithm is the depth first search subroutine of Ullman’s Algorithm. However, in our case the search space has been reduced to only the subgraphs that match the query. The sole verification that remains to be done concerns edge labels.

4 EXPERIMENTS

We evaluate the execution time performance of our algorithm, *SSI* (for Simple Subgraph Isomorphism), over various types of graphs and sizes of queries. We also compare it with one of the most efficient state of the art algorithm, called *TurboISO* and presented in [9]. *TurboISO* is itself compared to the other existing solutions in [9] and showed to be more efficient. We hoped to compare our algorithm to the work of [5] which is showed more efficient than *TurboISO* but we have not yet received any answer to our request from the authors.

All experiments are performed on a 2.40 GHz *Intel(R) Core(TM) i5* –

Algorithm 5: SubgraphList.

Data: a partial embedding M .

Result: All embeddings of Q in G .

```
begin
  if  $|M| = |V(Q)|$  then
    Report  $M$ ;
  end
  Choose a non matched vertex  $u$  from  $V(Q)$ ;
   $C_u \leftarrow \{ \text{non matched } v \in V_Q(G) \text{ such that } \text{FootMatch}(f(u), f(v)) \}$ ;
  foreach  $v \in C_u$  do
    if  $\text{neighborCheck}(u, v, M)$  then
       $M \leftarrow M \cup \{(u, v)\}$ ;
      SubgraphList( $M$ );
      Remove  $(u, v)$  from  $M$  ;
    end
  end
end
end
```

Algorithm 6: Function FootMatch.

Data: A query vertex u and a data vertex v .

Result: returns true if $f(u)$ matches $f(v)$.

```
begin
  return  $(f(u).label = f(v).label \wedge f(u).degree <$   

    $f(u).degree \wedge f(u).ne < f(v).ne)$  or  $(f(u).label =$   

    $f(v).label \wedge f(u).degree = f(u).degree \wedge f(u).ne = f(v).ne)$ 
end
```

Algorithm 7: Function neighborCheck.

Data: a partial embedding M , a query vertex u and a data vertex v .

Result: returns true if u and v have neighbors that match.

```
begin
  return  $\forall (u', v') \in M, ((u, u') \in E(Q) \rightarrow (v, v') \in E(G) \wedge \ell((u, u')) =$   

    $\ell((v, v')))$ 
end
```

4210U 64 bits laptop with 8 GB of RAM running windows 7. Algorithms are implemented in C++.

We first describe the datasets used in the experiments, then we present our results.

4.1 Datasets

We use nine datasets of real-world graphs to evaluate our approach. Table 1 summarises their characteristics. These datasets can be classified into three categories :

1. Small graphs: these graphs are known datasets used by almost all existing methods in their evaluation process. So, we mainly use them as comparative datasets. These datasets are referred to as AIDS, NASA, and HUMAN. The AIDS and HUMAN datasets are available in the RI database of biochemical data¹ [6]. The NASA dataset comes from the work of [12].
 - *AIDS database*: This dataset consists of graphs representing molecular compounds. It contains 10,000 graphs of 27 edges. The number of unique labels in AIDS is 51.
 - *NASA database*: This dataset contains 36,790 trees with an average size of 32, and a number of unique labels of 117,302.
 - *HUMAN*: This dataset consists of one large graph representing a protein interaction network. This graph has 4,675 vertices and 86,282 edges. The number of unique labels in the dataset is 90.

To query these three small datasets, we use queries constructed by [9] to evaluate Turbo_{ISO}. For NASA and AIDS, the authors of [9] constructed 6 query sets (Q4, Q8, Q12, Q16, Q20, Q24), each of which contains 1,000 query graphs of the same size. For HUMAN, the authors of [9] generated 10 query sets obtained by varying the number of query sizes from 1 to 10.

2. Large graphs: these graphs come from the Stanford Large Network Dataset Collection² and are referred to as Patent Citation, Pokec, LiveJournal and Orkut. They are large graphs but that still fit into main memory.
 - *Pokec* : Pokec is a highly popular on-line social network in Slovakia that contains friendship relationships.
 - *Patent Citation* : This is the citation network among US Patents. This dataset contains all the utility patents granted from January 1, 1963 to December 30, 1999. It includes almost 4 million patents and almost 17 millions citations.
 - *LiveJournal*: is an on-line social network with almost 5 million members and over 68 million friendship relations.

¹<http://ferrolab.dmi.unict.it/ri/ri.html#description>

²<http://snap.stanford.edu/>

- *Orkut*: is a free on-line social network with more than 3 million members and more than 117 friendship connections [1].
3. Big Graphs: these graphs come also from the Stanford Large Network Dataset Collection. They are Twitter2009 and Friendster. These graphs do not fit in main memory. To process these big graphs, we partitioned each disk file into several sequential files that fit into memory. Our algorithm relies on only sequential scans of disk files with a limited amount of main memory.
- Twitter2009: is a snapshot of the twitter microblogging social network that corresponds to the period of June-Dec 2009. The vertices represent users and edges correspond to user-follower relationships.
 - Friendster: Friendster [2] is an on-line social network where edges correspond to friendship relations. It contains more than 65 million vertices and more than 180 billion edges [3].

Table 1: Graph Dataset Characteristics. $avg|V|$: average number of vertices. $avg|E|$: average number of edges.

Dataset	Number of graphs	$avg V $	$avg E $
AIDS	10,000	26	27
NASA	36,790	94	32
HUMAN	1	4,675	86,282
PATENT CITATION	1	3,774,762	16,518,948
POKEC	1	1,632,803	30,622,564
LIVEJOURNAL	1	4,847,571	68,993,773
ORKUT	1	3,072,441	117,185,083
TWITTER2009	1	17,069,982	476,553,560
FRIENDSTER	1	65,608,366	180,606,731,005

For the large and big graphs, i.e., Kopec, Patent Citation, LiveJournal, Orkut, Twitter2009 and Friendster, we first added labels to the vertices and edges. These labels were generated randomly in the interval $[1, |V(G)|]$. Then, we constructed for each graph, 5 query sets (Q100, Q200, Q300, Q400, Q500). Each set Q_i contains 100 query graphs of the same size i .

The time performance reported in the results is the average time computed over the sets of queries of the same size.

For big graphs, i.e., Twitter2009 and Friendster, that do not fit into memory, the edges were sorted. So, all the filtering steps are applied during the file loading. This enable us to deals with these huge graphs even with non-distinguishing labels.

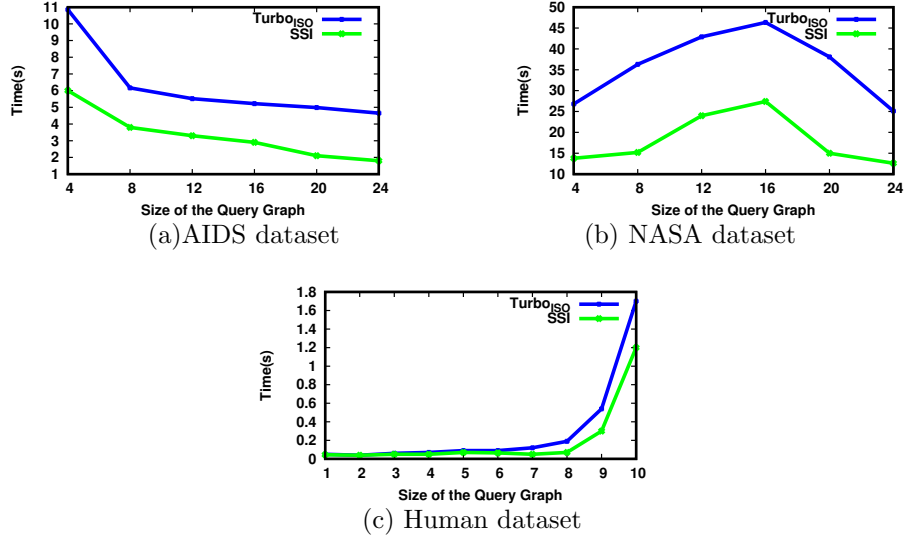


Figure 4: Experiments on small datasets.

4.2 Results

Figure 4 shows the experimental results for the small graphs for both *SSI* and Turbo_{ISO} [9]. We can clearly see that *SSI* outperforms Turbo_{ISO} for the three datasets. In fact, *SSI* considerably outperforms Turbo_{ISO} for NASA and AIDS and even with the HUMAN dataset which is particularly adapted to the vertex aggregation of Turbo_{ISO}, *SSI* is faster.

Figure 5 shows the total occurrences listing time for the two algorithms on the large graphs. The improvement brought by *SSI* is over 4 orders of magnitude for LiveJournal and Pokec and 2 orders magnitude for the two other datasets.

Figure 6 shows the total processing time of *SSI* on the two big graphs. It was not possible to run Turbo_{ISO} on these huge graphs without out-of memory errors. The processing time of *SSI* increases lineally with respect to the number of vertices in the query graph. These results definitely settle the effectiveness of the proposed approach.

5 Conclusions

Subgraph isomorphism search is an NP-complete problem. This means a processing time that grows with the size of the involved graphs. Pruning the search space is the pillar of a scalable subgraph isomorphism search algorithm and has been the main focus of proposed approaches since Ullman’s first solution. In this paper, we proposed *SSI*, a simple subgraph isomorphism search algorithm that reduces maximally intermediate results. Using a simple representation of the vertices, *SSI* distills topological information of each vertex into an integer.

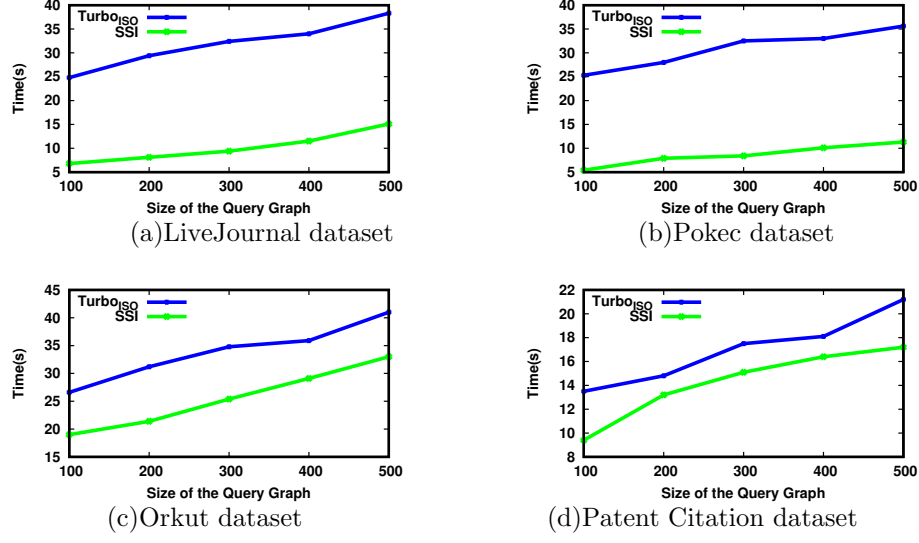


Figure 5: Results on large graphs.

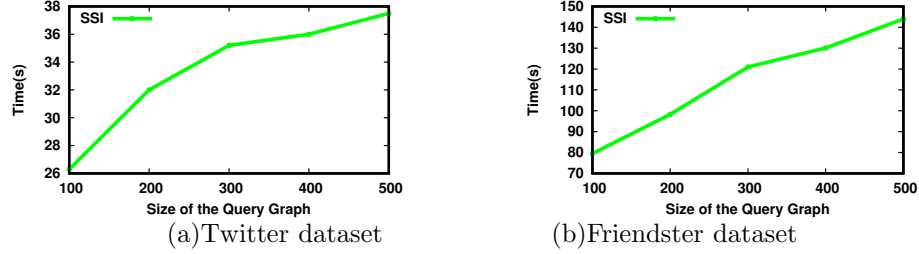


Figure 6: Results on big graphs.

This vertex encoding is simply updatable and can be used to prune globally the search space. Furthermore *SSI* does not require that the entire data graph is loaded into main memory and can be used with a graph stream. Our extensive experiments validate the efficiency of our approach. These results also open several research perspectives. First, it is interesting to extend this encoding method to embrace k -neighbourhood in order to enhance filtering in the first iterations of the filtering step of the algorithm. Second, it will be interesting to investigate how to make this encoding method computable incrementally without sorting the edges of the input graph. Third, we project to implement the CFL framework [5] if we do not receive a positive response from the authors. This will allow us to have more comparisons. Finally, extending the vertex encoding proposed in this paper to construct a graph index that allows to handle a graph database is an interesting issue as indexing is the first filtering step when we deal with a database of graphs. For this issue, we

conjecture that by computing a vertex encoding integer that includes the vertex label: $ne(u) = \sum_{j=1}^k h(j, x_1 + \dots + x_j)$ where the label of u is among the x_i and then compute a graph encoding integer using the same formula as follows: $ne(G) = \sum_{j=1}^k h(j, x_1 + \dots + x_j)$ where each x_i is the vertex encoding integer of a vertex of G . This resulting graph encoding integer can be used to index a graph in a database of graphs defined on the same set of labels. It can be also used as a comparison tool between graphs as it embeds label and topological data in the graph. This conjecture is under investigation for a theoretical proof and also an implementation.

References

- [1] <http://socialnetworks.mpi-sws.org/data-imc2007.html>.
- [2] <http://www.friendster.com/>.
- [3] <https://archive.org/details/friendster-dataset-201107>.
- [4] C. Bachman. Data structure diagrams. *DataBase: : A Quarterly Newsletter of SIGBDP*, 1(2), 1969.
- [5] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1199–1214, New York, NY, USA, 2016. ACM.
- [6] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics*, 14(Suppl 7)(S13), 2013.
- [7] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:1367–1372, 2004.
- [8] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS*, 6:45–53, 2006.
- [9] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 337–348, New York, NY, USA, 2013. ACM.
- [10] H. He and A. Singh. Closure-tree: An index structure for graph queries. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, pages 38–38, April 2006.
- [11] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 405–418, New York, NY, USA, 2008. ACM.

- [12] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 133–144. VLDB Endowment, 2013.
- [13] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, Aug. 2008.
- [14] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 39–52, New York, NY, USA, 2002. ACM.
- [15] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *J. ACM*, 23(1):31–42, Jan. 1976.
- [16] S. Zhang, S. Li, and J. Yang. Gaddi: Distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 192–203, New York, NY, USA, 2009. ACM.
- [17] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1):340–351, 2010.
- [18] X. Zhao, C. Xiao, X. Lin, and W. Wang. Efficient Graph Similarity Joins with Edit Distance Constraints. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April*, pages 834–845, 2012.

A Proof of Theorem 1

Proof. We need the following lemmas.

Lemma 6. $p < p' \Rightarrow h(k, p) < h(k, p')$

Proof. By deduction from the property of the binomial coefficient: $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ (Pascal Formula) \square

Lemma 7. $\forall k > 0, g_k(x_1, \dots, x_k) < h(k, x_1 + \dots + x_k + 1)$

Proof. This inequality is trivial for $k = 1$: $g_1(x_1) = x_1$ and $h(1, x_1 + 1) = x_1 + 1$. Assume that, for $k \geq 1$, the inequality holds and let us prove that it also holds for $k + 1$.

By definition of g_k , we have:

$$\begin{aligned}
 g_{k+1}(x_1, \dots, x_{k+1}) &= g_k(x_1, \dots, x_k) + h(k+1, x_1 + \dots + x_{k+1}) &< \\
 h(k, x_1 + \dots + x_k + 1) + h(k+1, x_1 + \dots + x_{k+1}) &< h(k, x_1 + \dots + x_k + 1) + h(k+1, x_1 + \dots + x_{k+1})
 \end{aligned}$$

By the property of Pascal's triangle, we know that:
 $\hbar(k, x_1 + \dots + x_k + x_{k+1} + 1) + \hbar(k+1, x_1 + \dots + x_{k+1}) = \hbar(k+1, x_1 + \dots + x_{k+1} + 1)$,
 we have $g_{k+1}(x_1, \dots, x_{k+1}) < \hbar(k+1, x_1 + \dots + x_{k+1} + 1)$ □

Lemma 8. $\forall k > 0$, If $g_k(x_1, \dots, x_k) = g_k(x'_1, \dots, x'_k)$ then $x_1 + \dots + x_k = x'_1 + \dots + x'_k$

Proof. Assume that $g_k(x_1, \dots, x_k) = g_k(x'_1, \dots, x'_k)$.
 According to Lemma 7, we have:

$\hbar(k, x_1 + \dots + x_k) < g_k(x_1, \dots, x_k) = g_k(x'_1, \dots, x'_k) < \hbar(k, x_1 + \dots + x_k + 1)$
 we obtain then: $\hbar(k, x_1 + \dots + x_k) < \hbar(k, x_1 + \dots + x_k + 1)$ According to Lemma 6, $\hbar(k, p)$ is strictly increasing. So, the inequality $x_1 + \dots + x_k \leq x'_1 + \dots + x'_k$ holds. Similarly, we prove the inverse inequality. This proves that $x_1 + \dots + x_k = x'_1 + \dots + x'_k$. □

To prove Theorem 3, we first prove that g_k is injective from \mathbb{N}^k to \mathbb{N} . It is trivial for $k = 1$. In fact, $g_1 = \hbar(1, x_1) = \binom{x_1}{1} = \frac{x_1!}{1!(x_1-1)!} = x_1$ is the identity in \mathbb{N} . For $k \geq 2$, we assume that g_{k-1} is injective and we prove that g_k is also injective. Let (x_1, \dots, x_k) and (x'_1, \dots, x'_k) such that $g_k(x_1, \dots, x_k) = g_k(x'_1, \dots, x'_k)$. According to Lemma 8, $x_1 + \dots + x_k = x'_1 + \dots + x'_k$. We have also by definition of g_k :

$$\begin{cases} g_k(x_1, \dots, x_k) = g_{k-1}(x_1, \dots, x_{k-1}) + \hbar(k, x_1 + \dots + x_k) \\ g_k(x'_1, \dots, x'_k) = g_{k-1}(x'_1, \dots, x'_{k-1}) + \hbar(k, x'_1 + \dots + x'_k) \end{cases}$$

By subtracting side by side, we obtain $g_{k-1}(x_1, \dots, x_{k-1}) = g_{k-1}(x'_1, \dots, x'_{k-1})$ which is our induction hypothesis that gives $(x_1, \dots, x_{k-1}) = (x'_1, \dots, x'_{k-1})$. This implies that $x_k = x'_k$.

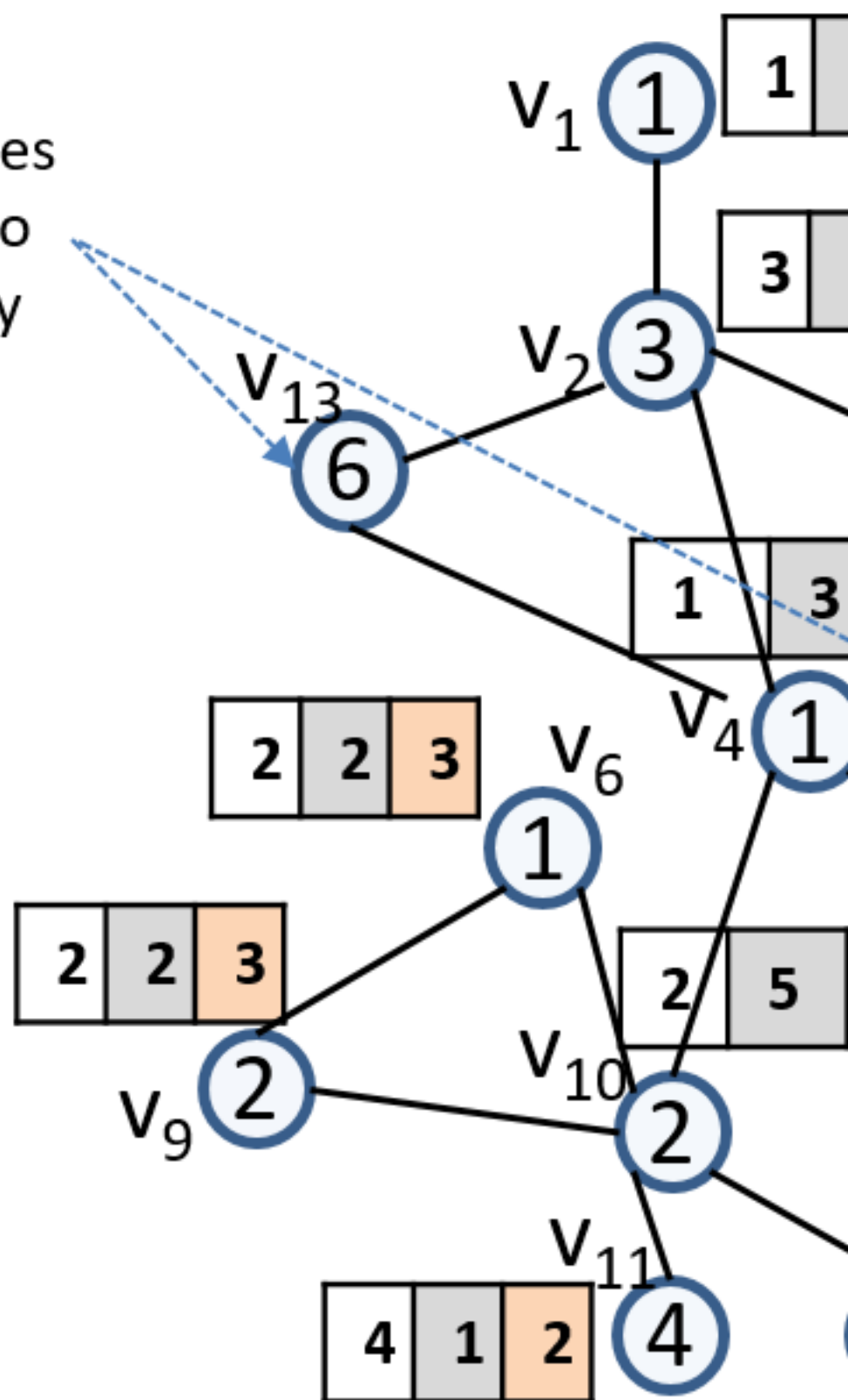
Conclusion: g_k is injective.

To show that g_k is also surjective, we recall that $\hbar(k, x_1 + \dots + x_k) \leq g_k(x_1, \dots, x_k) < \hbar(k, x_1 + \dots + x_k + 1)$. As $p \rightarrow \hbar(k, p)$ is a strictly increasing sequence, we deduce that each n in \mathbb{N} have an antecedent in \mathbb{N}^k .

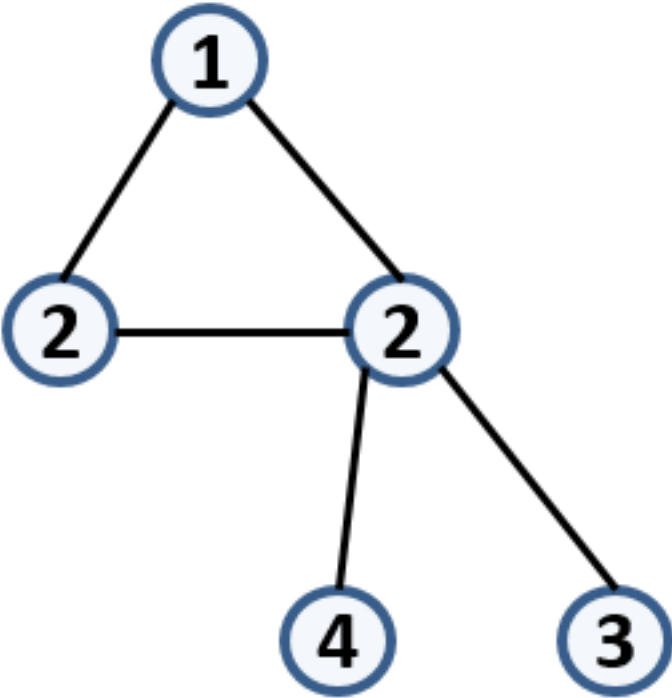
So, g_k is a bijection from \mathbb{N}^k to \mathbb{N} which proves Theorem 3. □

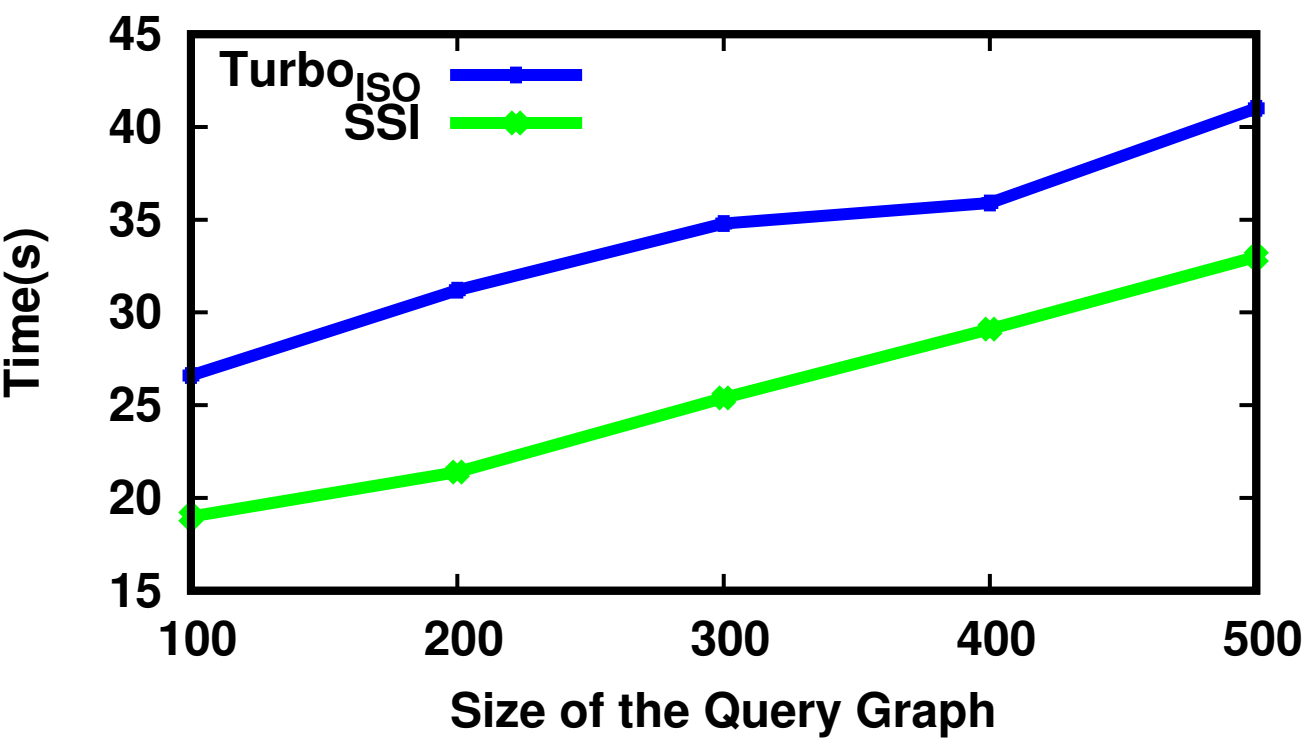
No footprint is
computed for vertices
 v_{13} and v_{14} as they do
not match any query
label

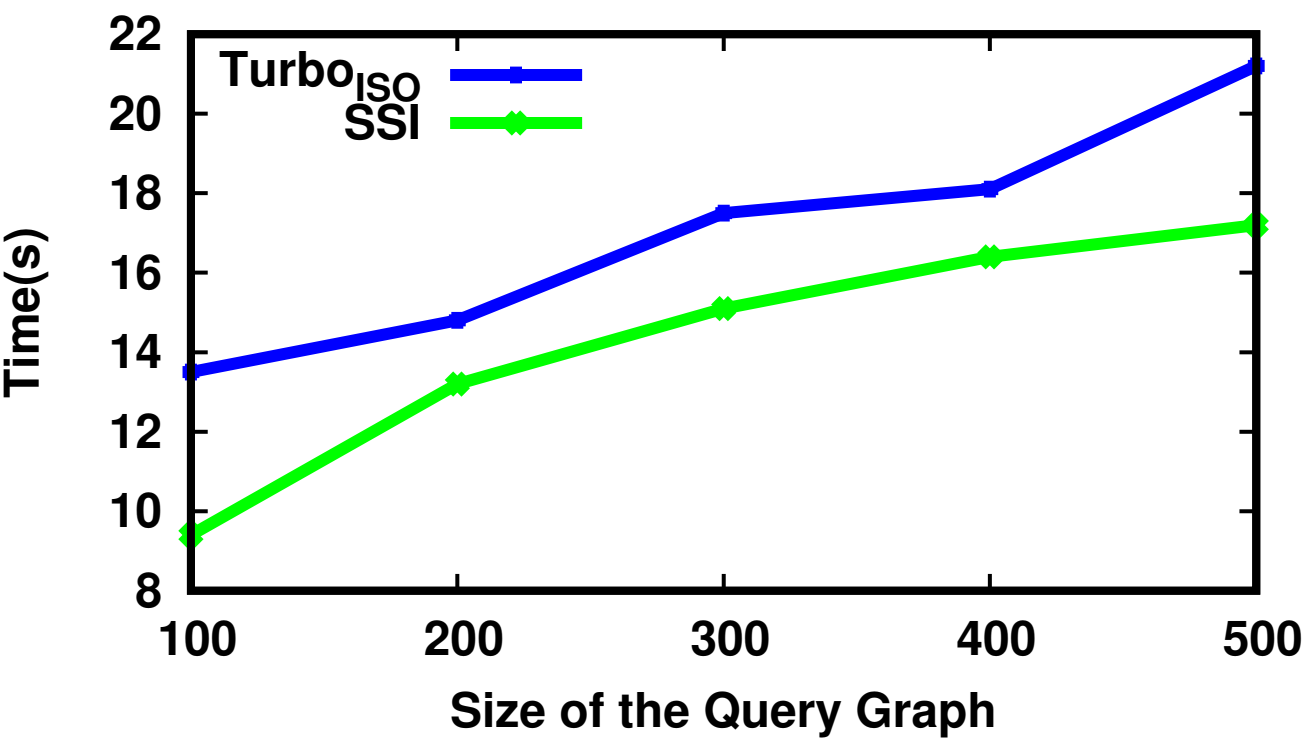
Data graph

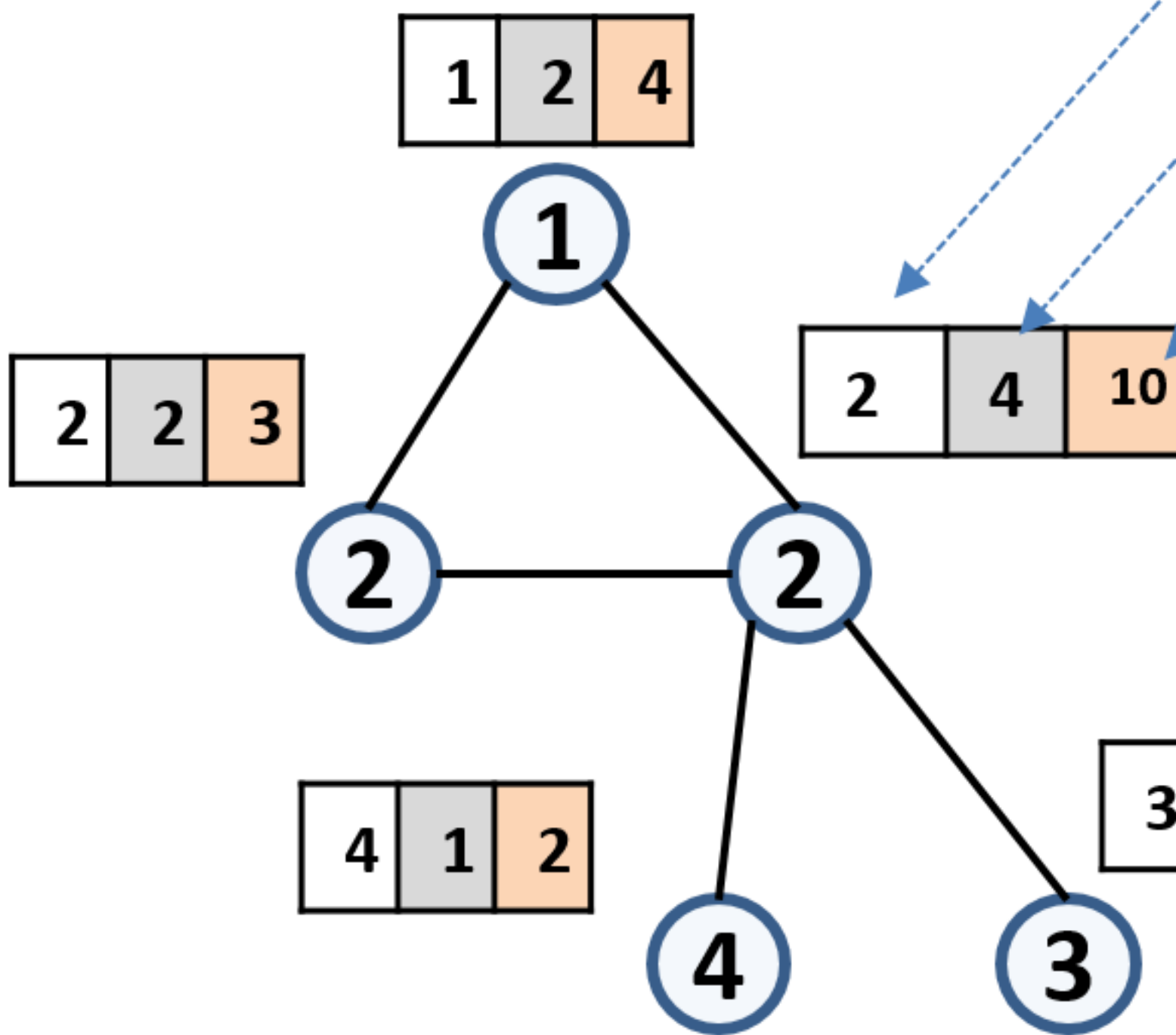


Query graph









Query graph

